

Diapo 1 - Les Design Patterns ou patrons de conception. Quelques pistes pour appliquer cette méthodologie dans des développements VFP.

Diapo 2 - Qui suis-je ?

Diapo 3 – Un peu de théorie

Diapo 4 - Un peu d'histoire

Les patrons de conception sont issus des travaux de l'architecte Christopher Alexander. Ils sont introduits progressivement en informatique à partir de 1987 et formalisés dans l'ouvrage d'Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides intitulé « Design Patterns – Elements of Reusable Object-Oriented Software » paru le 21-10-1994 avec un copyright de 1995. Les quatre auteurs sont souvent désignés par l'expression « Gang Of Four » (GOF) dans la littérature informatique. Cet ouvrage présente 23 Design Patterns et donne des exemples en C++ et en SmallTalk.

Il est traduit dès 1996 par Jean-Marie Lasvergères pour le compte des éditions International Thomson Publishing France sous le titre « Design patterns. Catalogue de modèles de conception réutilisables ». La dernière édition française date de 2007 aux éditions Vuibert. Les mauvaises langues indiquent des erreurs de traduction.

Diapo 5 - Bibliographie

Petite remarque concernant la bibliographie. J'ai été surpris, lors de mes recherches, de l'indisponibilité chez les éditeurs de la plupart des ouvrages en français cités dans les différentes ressources que j'ai utilisées pour rédiger cet exposé.

Diapo 6 - Qu'est-ce qu'un patron de conception ?

« Chaque modèle décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois, sans le faire deux fois de la même manière. » (Christopher Alexander)

Dans le domaine de l'architecture des logiciels un patron de conception représente la description d'une solution à un problème récurrent de conception. Il relève de ce que l'on appelle « les bonnes pratiques » de développement.

Diapo 7 - Un patron de conception est un ensemble de relations et de règles liant plusieurs interfaces ou classes qui expriment la solution à un problème donné dans un contexte précis.

Les patrons de conception améliorent la qualité et la fiabilité du développement en minimisant les couplages, ou relations de dépendance, au sein d'une application. Ils permettent également de renforcer la cohésion au sein des ensembles de classes reliées fonctionnellement.

Ceux qui maintiennent une grosse application connaissent l'effet « spaghetti » (pour les voraces) ou « dominos » (pour les mécaniciens). Les classes manipulées sont tellement dépendantes les unes des autres qu'une modification de l'une d'entre elles peut mettre la stabilité de l'ensemble de l'édifice en difficulté. Les patrons de conception permettent de diminuer ces effets de bords.

Attention cependant à ne pas les instituer en modèles absolus. Inutile de compliquer une petite application par l'utilisation d'une méthodologie conçue pour des applications lourdes et complexes.

En résumé.

Diapo 8 - Ce que sont les patrons de conception :

- la description d'une solution à un problème récurrent de conception
- la description d'une partie de la solution, les autres parties étant les environnements logiciels et matériels
- une technique d'architecture logicielle

Diapo 9 - Ce que ne sont pas les patrons de conception :

- une brique logicielle (le patron dépend de son contexte)
- une règle que l'on peut appliquer mécaniquement (parce que justement il dépend de son contexte)
- une méthode de prise de décision, le patron est la décision.

Comment décrire les patrons de conception ?

Diapo 10 - Les quatre éléments d'un modèle sont :

- le nom du modèle
- le problème dont le modèle est une solution (ou intention)
- la solution
- les conséquences

Diapo 11 - Un patron de conception peut avoir

- soit une portée de classe en se focalisant sur les relations entre les classes et leurs sous-classes, le mécanisme utilisé étant l'héritage
- soit une portée d'objet (ou d'instance) en se focalisant sur les relations entre les objets, le mécanisme utilisé étant la composition

Diapo 12 - Il existe trois catégories de patrons de conception :

- créationnels (mécanismes pour l'instanciation d'objets)
- structuraux (organisation des interfaces et des classes en utilisant les mécanismes d'agrégation et de composition)
- comportementaux (communication entre les classes et répartition des responsabilités)

Diapo 13 - Les 23 patrons de conception

Diapo 14 - Bref retour sur les interfaces.

Il sera beaucoup question d'interface lors de cet exposé. Les utilisateurs de langage tels que Java ou C# sont très certainement familiers de cette notion. D'autres, fins connaisseurs du C++, préféreront parler d'héritage multiple. Tout le monde se retrouvera sur les classes abstraites.

Malheureusement VFP ne connaît aucune de ces notions. Il est toujours possible d'assumer les contraintes liées à l'interface à travers des règles de programmation et/ou des commentaires appropriés.

De quoi s'agit-il ? Un objet peut hériter ses principaux comportements d'un autre objet mais également être amené à respecter les comportements d'un ou plusieurs autres objets. Pour formaliser ce type de contraintes on introduit la notion de contrat qu'un objet s'engage à respecter. Techniquement parlant, on implémente un contrat par la définition d'une interface.

Un contrat est constitué par l'ensemble des messages qu'un objet est capable de traiter. Chaque message est caractérisé par une méthode (son nom), des paramètres (leurs types) et une réponse (type de retour). Ce sont ces trois constituants - nom, types des paramètres, type de retour - qui constituent la signature d'une méthode.

Un contrat est ainsi constitué par un ensemble de signatures qui doivent être toutes assumées. Un objet peut respecter plusieurs contrats.

Sous VFP on pourra considérer que l'interface d'une classe est l'ensemble de ses méthodes publiques.

Diapo 15 - Interfaces, classes, implémentations, objets

Les patrons de conception sont étroitement liés aux langages de programmation orientés objets. Ils accordent une place de premier plan aux interfaces. « Penser interface plutôt qu'implémentation » telle pourrait être leur devise.

Les classes et fonctions abstraites sont utilisées pour définir les interfaces communes à un ensemble de classes.

L'utilisation d'un objet doit privilégier sa vision comme étant celle d'une instance de la classe abstraite dont sa classe concrète hérite. Ainsi l'on rend indépendantes les interfaces de leurs implémentations et les objets utilisateurs des objets utilisés. Ainsi faut-il déclarer les paramètres-objets comme instances des classes abstraites dont ils héritent plutôt que comme instances des classes particulières qui ont procédé à leur construction.

Remarques concernant l'héritage, l'agrégation et la composition.

Pour qu'une classe hérite du comportement d'une autre, trois méthodes sont disponibles.

Diapo 16 - La première, la plus connue, est l'héritage. On parle aussi de spécialisation de classe. C'est de loin la méthode la plus simple si une seule interface est assumée par un objet. Attention cependant à ne pas faire dégénérer la hiérarchie des classes en un fouillis d'objets hyperspécialisés.

Un exemple (simplissime) pour illustrer le point précédent. Je désire sauvegarder les tailles et positions de certains de mes formulaires.

Diapo 17 - Héritage

```
*****
* Heritage *
*****

LOCAL loForm

loForm=CREATEOBJECT('myForm1')
loForm.SHOW()
READ EVENTS

DEFINE CLASS myForm1 AS FORM

    PROCEDURE INIT
        THIS.onRestore()
    ENDPROC

    PROCEDURE MOVED()
        THIS.onSave()
    ENDPROC

    PROCEDURE RESIZE()
        THIS.onSave()
    ENDPROC

    PROCEDURE DESTROY()
        CLEAR EVENTS
    ENDPROC

    PROCEDURE onSave

        LOCAL lnParam

        lnParam=STR(THIS.TOP,10)+STR(THIS.LEFT,10)+STR(THIS.WIDTH,10)+STR(THIS.HEIGHT,10)
        STRTOFILE(lnParam,'C:\Temp\'+THIS.NAME+'.txt',0)

    ENDPROC

    PROCEDURE onRestore
```

```

LOCAL lnParam

IF FILE('C:\Temp\'+THIS.NAME+'.txt')
    lnParam=FILETOSTR('C:\Temp\'+THIS.NAME+'.txt')
    THIS.TOP=INT(VAL(SUBSTR(lnParam,1,10)))
    THIS.LEFT=INT(VAL(SUBSTR(lnParam,11,10)))
    THIS.WIDTH=INT(VAL(SUBSTR(lnParam,21,10)))
    THIS.HEIGHT=INT(VAL(SUBSTR(lnParam,31,10)))
ENDIF

ENDPROC

ENDDDEFINE

```

Diapo 18 - La deuxième méthode est l'agrégation. Elle consiste à définir une propriété comme étant une référence vers l'instance d'un objet dont on doit assumer l'interface. Elle permet de simuler l'héritage multiple. Plus modulaire que la spécialisation elle permet d'adapter finement le comportement à l'exécution par le choix au dernier moment de la classe réelle de l'objet instancié.

Diapo 19 - Agrégation

```

*****
* Agregation *
*****

LOCAL loForm

loParam=CREATEOBJECT('oParam')
loForm=CREATEOBJECT('myForm3',loParam)
loForm.SHOW()
READ EVENTS
RELEASE loForm
RELEASE loParam

DEFINE CLASS myForm3 AS FORM

    loParam=NULL

    PROCEDURE INIT(poParam)
        THIS.loParam=poParam
        THIS.loParam.onRestore(THIS)
    ENDPROC

    PROCEDURE MOVED()
        THIS.loParam.onSave(THIS)
    ENDPROC

    PROCEDURE RESIZE()
        THIS.loParam.onSave(THIS)
    ENDPROC

    PROCEDURE DESTROY()
        CLEAR EVENTS
    ENDPROC

ENDDDEFINE

DEFINE CLASS oParam AS CUSTOM

    PROCEDURE onSave(poForm)

        LOCAL lnParam

        lnParam=STR(poForm.TOP,10)+STR(poForm.LEFT,10)+STR(poForm.WIDTH,10)+STR(poForm.HEIGHT,10)
        STRTOFILE(lnParam,'C:\Temp\'+poForm.NAME+'.txt',0)

    ENDPROC

    PROCEDURE onRestore(poForm)

        LOCAL lnParam

        IF FILE('C:\Temp\'+poForm.NAME+'.txt')
            lnParam=FILETOSTR('C:\Temp\'+poForm.NAME+'.txt')
            poForm.TOP=INT(VAL(SUBSTR(lnParam,1,10)))
            poForm.LEFT=INT(VAL(SUBSTR(lnParam,11,10)))
            poForm.WIDTH=INT(VAL(SUBSTR(lnParam,21,10)))
            poForm.HEIGHT=INT(VAL(SUBSTR(lnParam,31,10)))
        ENDIF
    ENDPROC

```

```
ENDPROC
```

```
ENDDDEFINE
```

Diapo 20 - La troisième méthode est la composition. Identique dans son principe à l'agrégation, elle se différencie de celle-ci par le fait qu'elle contrôle complètement le cycle de vie de l'objet instancié. Celui-ci est créé pour un usage exclusif. Il sera détruit lors de la destruction de l'objet hôte.

Diapo 21 - Composition

```
*****  
* Composition *  
*****
```

```
LOCAL loForm
```

```
loForm=CREATEOBJECT('myForm2')  
loForm.SHOW()  
RELEASE loForm
```

```
DEFINE CLASS myForm2 AS FORM
```

```
    loParam=NULL
```

```
    PROCEDURE INIT
```

```
        THIS.loParam=CREATEOBJECT('oParam', THIS)
```

```
        THIS.loParam.onRestore()
```

```
    ENDPROC
```

```
    PROCEDURE MOVED()
```

```
        THIS.loParam.onSave()
```

```
    ENDPROC
```

```
    PROCEDURE RESIZE()
```

```
        THIS.loParam.onSave()
```

```
    ENDPROC
```

```
ENDDDEFINE
```

```
DEFINE CLASS oParam AS CUSTOM
```

```
    loForm=NULL
```

```
    PROCEDURE INIT(poForm AS FORM)
```

```
        THIS.loForm=poForm
```

```
    ENDPROC
```

```
    PROCEDURE onSave
```

```
        LOCAL lnParam
```

```
        lnParam=STR(THIS.loForm.TOP,10)+STR(THIS.loForm.LEFT,10)+STR(THIS.loForm.WIDTH,10)+STR(THIS.loForm.HEIGHT,10)  
        STRTOFILE(lnParam,'C:\Temp\'+THIS.loForm.NAME+'.txt',0)
```

```
    ENDPROC
```

```
    PROCEDURE onRestore
```

```
        LOCAL lnParam
```

```
        IF FILE('C:\Temp\'+THIS.loForm.NAME+'.txt')
```

```
            lnParam=FILETOSTR('C:\Temp\'+THIS.loForm.NAME+'.txt')
```

```
            THIS.loForm.TOP=INT(VAL(SUBSTR(lnParam,1,10)))
```

```
            THIS.loForm.LEFT=INT(VAL(SUBSTR(lnParam,11,10)))
```

```
            THIS.loForm.WIDTH=INT(VAL(SUBSTR(lnParam,21,10)))
```

```
            THIS.loForm.HEIGHT=INT(VAL(SUBSTR(lnParam,31,10)))
```

```
        ENDIF
```

```
    ENDPROC
```

```
ENDDDEFINE
```

J'ai préféré l'écriture

```
    THIS.loParam=CREATEOBJECT('oParam', THIS)
```

à l'écriture

```
    THIS.ADDOBJECT('loParam', 'oParam')
```

car elle permet d'indiquer clairement l'utilisation d'une propriété déléguée pour l'implémentation d'une interface.

Diapo 22 - Les auteurs les plus sérieux affirment qu'il faut préférer la composition d'objets à l'héritage. Les dépendances d'implémentation sont ainsi évitées. La programmation est centrée sur la réalisation d'une tâche et les comportements peuvent changer en cours d'exécution.

Dans la littérature les notions d'agrégation et de composition sont variables.

Attention : dans l'ouvrage du Gof, l'agrégation désigne la composition et l'accointance désigne l'agrégation.

Diapo 23 – Un peu de pratique

Diapo 24 - Adaptateur

« Convertit l'interface d'une classe en une autre conforme à l'attente du client. L'adaptateur (adapter) permet à des classes de collaborer qui n'auraient pu le faire du fait d'interfaces incompatibles. »

Diapo 25 - Pour exemple, je vais prendre la fonction ShellExecute que j'ai empaqueté dans une classe WinApi. La fonction onShellExecute admet les mêmes paramètres que l'API ShellExecute et effectue un certain nombre de vérification sur ceux-ci. La signature de cette fonction est **INT onShellExecute STRING STRING STRING STRING INT**

```
DEFINE CLASS WinApi AS CUSTOM

    PROCEDURE INIT
        DECLARE INTEGER ShellExecute ;
            IN SHELL32.DLL ;
            INTEGER nWinHandle,;
            STRING cOperation,;
            STRING cFileName,;
            STRING cParameters,;
            STRING cDirectory,;
            INTEGER nShowWindow
    ENDPROC

    FUNCTION onShellExecute(tcFileName,tcWorkDir,tcOperation,tcParam,tnWindow)

        LOCAL lcFileName,lcWorkDir,lcOperation,lcParam,lnWindow

        IF EMPTY(tcFileName)
            RETURN -1
        ENDIF

        lcFileName=ALLTRIM(tcFileName)
        lcWorkDir=IIF(TYPE('tcWorkDir')='C',ALLTRIM(tcWorkDir),'')
        lcOperation=IIF(TYPE('tcOperation')='C' AND NOT
EMPTY(tcOperation),ALLTRIM(tcOperation),'Open')
        lcParam=IIF(TYPE('tcParam')='C',ALLTRIM(tcParam),'')
        lnWindow=IIF(TYPE('tnWindow')='N',INT(tnWindow),1)

        RETURN ShellExecute(0,lcOperation,lcFileName,lcParam,lcWorkDir,lnWindow)

    ENDFUNC

ENDEDEFINE
```

Diapo 26 - L'objectif est d'adapter la classe WinApi afin de répondre aux besoins de l'interface suivante :

```
DEFINE CLASS IDocument AS CUSTOM

    FUNCTION DocOpen(docName AS STRING) AS INTEGER
        RETURN -2
    ENDFUNC

    FUNCTION DocPrint(docName AS STRING) AS INTEGER
        RETURN -2
    ENDFUNC

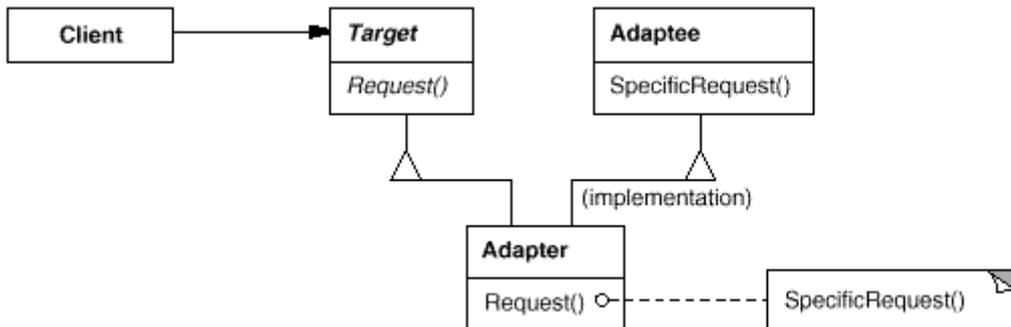
ENDEDEFINE
```

Diapo 27 – Les diagrammes utilisés sont ceux de l'ouvrage de référence. Ce sont des diagrammes de classe OMT (Object Modeling Technique), ancêtre d'UML (Unified Modeling Language)

Il existe deux formes d'implémentation du pattern adaptateur.

L'adaptateur de classe.

On spécialise la classe à adapter par héritage. On implémente ensuite l'interface à adapter. Pour cela on explicite les fonctions de l'interface cible en utilisant les méthodes de la classe à adapter.



Cette forme est la plus souple, surtout si les méthodes de la classe de base doivent être surchargées. Mais elle ne fonctionnera pas dans le cas où la classe adaptée est une sous-classe de la classe à adapter. Par exemple, dans le cas particulier des API, deux sous-classes WinApi32 et WinApi64 qui gèrent les deux plateformes disponibles sur la même machine. Lors de la spécialisation nous serons obligés de choisir l'une des deux formules, soit le 32 bits, soit le 64.

Diapo 28 – Exemple d'adaptateur de classe

```
DEFINE CLASS AdapterWinApi1 AS WinApi && implémente IDocument

    FUNCTION DocOpen(docName AS STRING) AS INTEGER
        RETURN THIS.onShellExecute(docName, '', 'open', '', 1)
    ENDFUNC

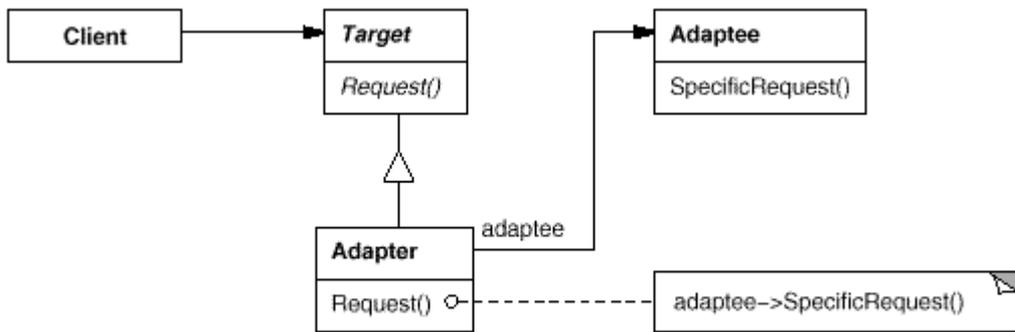
    FUNCTION DocPrint(docName AS STRING) AS INTEGER
        RETURN THIS.onShellExecute(docName, '', 'print', '', 1)
    ENDFUNC

ENDDFINE

oClient1=CREATEOBJECT('AdapterWinApi1')
oClient1.DocOpen('C:\AtoutFox\Lille2014\VFP\config9.txt')
oClient1.DocPrint('C:\AtoutFox\Lille2014\VFP\config9.txt')
RELEASE oClient1
```

Diapo 29 – L'adaptateur d'objet.

On incorpore à la classe adaptateur une instance de la classe à adapter. L'intérêt de cette méthode est de pouvoir fonctionner avec une sous-classe de la classe à adapter (en respectant l'interface commune aux sous-classes). L'inconvénient est la difficulté à redéfinir les comportements de la classe adaptée.



Diapo 30 – Exemple d'adaptateur d'objet

```
DEFINE CLASS AdapterWinApi2 AS CUSTOM && implémente IDocument
```

```
  loWinApi=NULL
```

```
  PROCEDURE INIT
```

```
    THIS.loWinApi=CREATEOBJECT('WinApi')
```

```
  ENDPROC
```

```
  FUNCTION DocOpen(docName AS STRING) AS INTEGER
```

```
    RETURN THIS.loWinApi.onShellExecute(docName,'','open','',1)
```

```
  ENDFUNC
```

```
  FUNCTION DocPrint(docName AS STRING) AS INTEGER
```

```
    RETURN THIS.loWinApi.onShellExecute(docName,'','print','',1)
```

```
  ENDFUNC
```

```
ENDEFINE
```

```
oClient2=CREATEOBJECT('AdapterWinApi2')
```

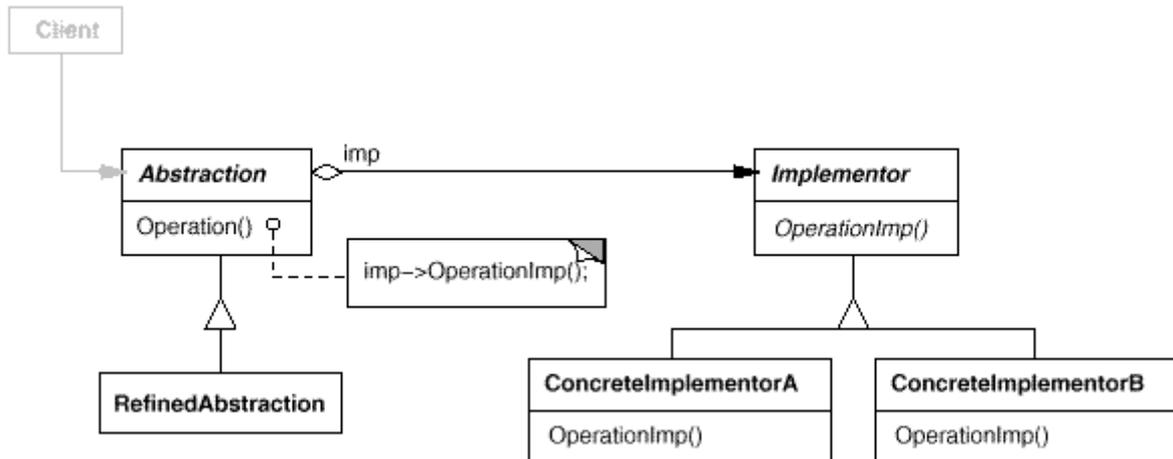
```
oClient2.DocOpen('C:\AtoutFox\Lille2014\VFP\config9.txt')
```

```
oClient2.DocPrint('C:\AtoutFox\Lille2014\VFP\config9.txt')
```

```
RELEASE oClient1
```

Diapo 31 - Pont

« Découple une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre. »



Diapo 32 - Pour exemple je vais utiliser une procédure d'impression d'un document commercial. Je définis donc une interface IDoc avec une seule méthode publique DocPrint.

L'analyse conduit à séparer le processus en cinq étapes : initialisation, impression de l'entête du document, impression du corps du document, impression du pied du document, finalisation, impression d'une ligne.

Diapo 33- Je peux définir mes classes comme suit :

```
DEFINE CLASS IDoc AS CUSTOM

    PROCEDURE DocPrint()
    ENDPROC

ENDEFINE

DEFINE CLASS Document1 AS IDoc

    PROTECTED cTypeDoc
    cTypeDoc='DOC'

    PROCEDURE DocPrint()
        THIS.DocPrintStart()
        THIS.DocPrintEntete()
        THIS.DocPrintCorps()
        THIS.DocPrintPied()
        THIS.DocPrintEnd()
    ENDPROC

    PROTECTED PROCEDURE DocPrintStart()
        THIS.DocPrintLine('')
        THIS.DocPrintLine("Sélection de l'imprimante")
        THIS.DocPrintLine('')
    ENDPROC

    PROTECTED PROCEDURE DocPrintEntete()
        THIS.DocPrintLine('Adresse1')
        THIS.DocPrintLine('Adresse2')
        THIS.DocPrintLine('CP Ville')
    ENDPROC

    PROTECTED PROCEDURE DocPrintCorps()
        THIS.DocPrintLine('')
        THIS.DocPrintLine('Article un')
        THIS.DocPrintLine('Article deux')
        THIS.DocPrintLine('')
    ENDPROC

    PROTECTED PROCEDURE DocPrintPied()
        THIS.DocPrintLine('Montant TTC')
```

```

ENDPROC

PROTECTED PROCEDURE DocPrintEnd()
    THIS.DocPrintLine('')
    THIS.DocPrintLine("Fermeture de l'imprimante")
    THIS.DocPrintLine('')
ENDPROC

PROTECTED PROCEDURE DocPrintLine(strLigne AS STRING)
    ? THIS.cTypeDoc+' '+strLigne
ENDPROC

ENDDDEFINE

```

Diapo 34 - Je veux ensuite spécialiser mon interface IDoc pour gérer deux types de document :

- des devis avec l'impression d'une lettre de présentation
- des factures avec l'impression des coordonnées bancaires

Diapo 35 – Puis, quelques mois plus tard, il me faut proposer trois types de sortie :

- imprimante
- PDF
- HTML

Diapo 36 – La première méthode pour traiter ces spécialisations consiste à tester le type de document et le type de sortie dans chacune des procédures ou fonctions affectées (bridge1.prg).

Diapos 37-39 –Le code en devient vite illisible. Les tests concernant le type de document peuvent cohabiter avec des tests concernant le type de sortie. Si un nouveau type de document est nécessaire, des parties très éloignées du code devront être modifiées.

Diapos 40 - 41 – La deuxième méthode est la spécialisation de la classe de départ. Attention à la prolifération des classes (bridge2.prg).

- **Diapo 42** - création de deux sous-classes pour traiter les différents types de document
- **Diapo 43** - création de six sous-classes pour traiter les différents types de sortie

Une troisième piste peut être explorée qui combine les deux précédentes méthodes : spécialisation sur le type de document et tests sur le type de sortie. Ou réciproquement. Elle combine les inconvénients des deux premières méthodes. On l'oubliera donc.

Diapo 44 – On voit dans cet exemple deux logiques à l'œuvre. L'une concerne les différentes phases d'impression d'un document. L'autre des opérations plus techniques qui sont conditionnées par le type de sortie.

Diapos 45 - 47 – Le patron de conception « pont » propose de découpler les deux interfaces : IDoc et sa méthode publique DocPrint et IPrint avec, pour l'instant, une seule méthode PrintLine. L'analyse montre que deux autres méthodes sont nécessaires : PrintStart et PrintEnd.

Dans les objets IDoc nous créons un objet dont l'interface est IPrint. Ainsi les méthodes privées de IDoc pourront utiliser les méthodes proposées par l'interface IPrint.

Les deux interfaces peuvent maintenant être spécialisées de manière indépendante.

Diapo 48 – En guise de conclusion

Trouver les bons objets.

Ils proposent des abstractions pour des objets qui n'apparaissent pas de manière naturelle : Composite, Strategy et State.

Bien choisir la granularité.

Ils permettent de choisir ce qui doit être regroupé ou ce qui doit être décomposé : Facade, Flyweight, Abstract Factory, Builder.

Penser « interface ».

Ils permettent d'externaliser ce qui est du domaine de l'interface : Memento, Decorator, Proxy, Visitor, Facade.

Spécifier l'implémentation

Ils ouvrent la voie à la modularité et au polymorphisme : Chain of responsibility (une seule interface pour des implémentations différentes), Composite (une interface avec une partie de l'implémentation dans le composite), Command, Observer, State, Strategy (souvent de pures interfaces abstraites), Prototype, Singleton, Factory, Builder (création d'objets satisfaisant une interface avec des implémentations diverses).

Favoriser la réutilisation.

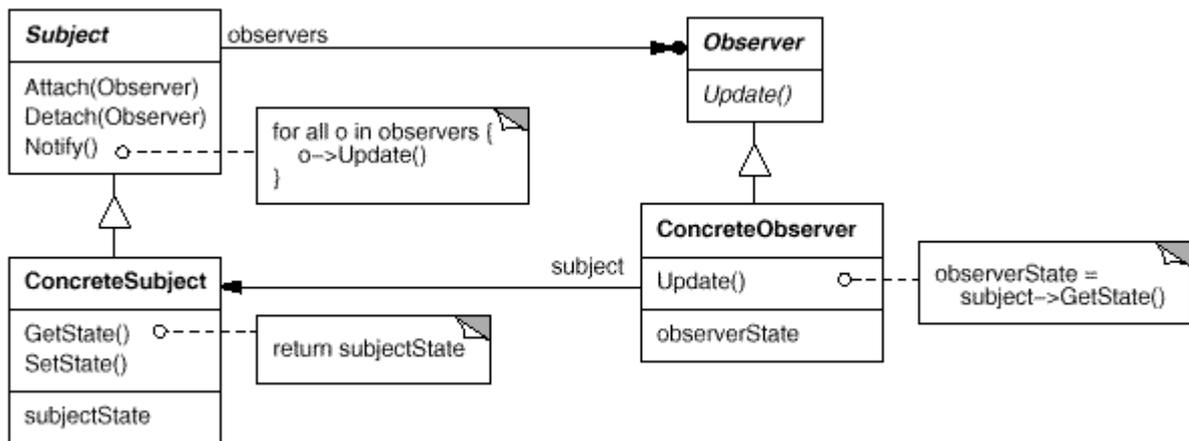
Préférer la composition (flexible et dynamique) à l'héritage (qui rompt en partie l'encapsulation).

Utiliser la délégation (forme de composition qui peut remplacer l'héritage) : Bridge, Mediator, Visitor, Proxy.

Bonus 1

La diapo de réserve pour de nouvelles aventures - Observateur

« Définit une interdépendance de type un à plusieurs, de façon que quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour »



Bonus 2

Les patrons de conception du GoF

Créationnels (Creational) - classe

Fabrication (Factory Method)

Définit une interface pour la création d'un objet, tout en laissant à des sous-classes le choix de la classe à instancier. Une fabrication permet de déléguer à des sous-classes les instanciations d'une classe.

Créationnels (Creational) - objet

Fabrique abstraite (Abstract Factory)

Fournit une interface, pour créer des familles d'objets apparentés ou dépendants, sans avoir à spécifier leurs classes concrètes.

Monteur (Builder)

Dans un objet complexe, dissocie sa construction de sa représentation, de sorte que, le même procédé de construction puisse engendrer des représentations différentes.

Prototype (Prototype)

Spécifie les espèces d'objets à créer, en utilisant une instance de type prototype, et crée de nouveaux objets par copies de ce prototype.

Singleton (Singleton)

Garantit qu'une classe n'a qu'une seule instance, et fournit à celle-ci, un point d'accès de type global.

Structuraux (Structural) - classe

Adaptateur (Adapter)

Convertit l'interface d'une classe en une interface distincte, conforme à l'attente de l'utilisateur. L'adaptateur permet à des classes de travailler ensemble, qui n'auraient pu le faire autrement pour cause d'interfaces incompatibles.

Structuraux (Structural) - objet

Adaptateur (Adapter)

Convertit l'interface d'une classe en une interface distincte, conforme à l'attente de l'utilisateur. L'adaptateur permet à des classes de travailler ensemble, qui n'auraient pu le faire autrement pour cause d'interfaces incompatibles.

Pont (Bridge)

Découple une abstraction de son implémentation associée, afin que les deux puissent être modifiées indépendamment.

Composite (Composite)

Organise les objets en structure arborescente représentant la hiérarchie de bas en haut. Le composite permet aux utilisateurs de traiter des objets individuels, et des ensembles organisés de ces objets de la même façon.

Décorateur (Decorator)

Attache des responsabilités supplémentaires à un objet de façon dynamique. Il permet une solution alternative pratique pour l'extension des fonctionnalités, à celle de dérivation de classes.

Façade (Facade)

Fournit une interface unifiée pour un ensemble d'interfaces d'un sous-système. Façade définit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.

Poids Mouche (Flyweight)

Assure en mode partagé le support efficace d'un grand nombre d'objets à fine granularité.

Procuration (Proxy)

Fournit un subrogé ou un remplaçant d'un autre type d'objet, pour en contrôler l'accès.

Comportementaux (Behavioral) -classe

Interprète (Interpreter)

Dans un langage donné, il définit une représentation de sa grammaire, ainsi qu'un interprète qui utilise cette représentation pour analyser la syntaxe du langage.

Patron de méthode (Template Method)

Définit le squelette de l'algorithme d'une opération, en déléguant le traitement de certaines étapes à des sous-classes. Le patron de méthode permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de l'algorithme.

Comportementaux (Behavioral) -objet

Chaîne de responsabilité (Chain of responsibility)

Permet d'éviter de coupler l'expéditeur d'une requête à son destinataire, en donnant la possibilité à plusieurs objets de prendre en charge la requête. Pour ce faire, il chaîne les objets récepteurs, et fait passer la requête tout au long de cette chaîne jusqu'à ce que l'un des objets la prenne en charge.

Commande (Command)

Encapsule une requête comme un objet, ce qui permet de faire un paramétrage des clients avec différentes requêtes, files d'attente, ou historiques de requêtes, et d'assurer le traitement des opérations réversibles.

Itérateur (Iterator)

Fournit un moyen pour accéder en séquence aux éléments d'un objet de type agrégat sans révéler sa représentation sous-jacente.

Médiateur (Mediator)

Définit un objet qui encapsule les modalités d'interaction de divers objets. Le médiateur favorise les couplages faibles, en dispensant les objets d'avoir à faire référence explicite les uns aux autres ; de plus, il permet de modifier une relation indépendamment des autres.

Memento (Memento)

Sans violer l'encapsulation, acquiert et délivre à l'extérieur une information sur l'état interne d'un objet, afin que celui-ci puisse être rétabli ultérieurement dans cet état.

Observateur (Observer)

Définit une corrélation entre objets du type un à plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent, en soient notifiés et mis à jour automatiquement.

Etat (State)

Permet à un objet de modifier son comportement lorsque son état interne change. L'objet paraîtra changer de classe.

Stratégie (Strategy)

Définit une famille d'algorithmes, encapsule chacun d'entre eux, et les rend interchangeables. Une stratégie permet de modifier un algorithme indépendamment de ses clients.

Visiteur (Visitor)

Représente une opération à effectuer sur les éléments d'une structure d'objets. Le visiteur permet de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère.

Bonus 3

Les liens que j'ai visité pour construire cette session

<http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/>
<http://abrillant.developpez.com/tutoriel/java/design/pattern/introduction/>
<http://www.emse.fr/~picard/cours/2A/DesignPatternsISI.pdf>
<http://dept-info.labri.u-bordeaux.fr/~griffaul/Enseignement/POO/Cours/cours/>
<http://design-patterns.fr/>
<http://www.goprod.bouhours.net/?page=accueil>
<http://www.jmdoudoux.fr/java/dej/chap-design-patterns.htm>

Après j'ai arrêté les recherches sur la toile et je me suis plongé dans la lecture (papier) de la traduction française de l'ouvrage du GoF.